HAL
archives-ouvertes.fr

A Polynomial Spilling Heuristic: Layered Allocation

Boubacar Diouf, Albert Cohen, Fabrice Rastello

▶ **To cite this version:**

Boubacar Diouf, Albert Cohen, Fabrice Rastello. A Polynomial Spilling Heuristic: Layered Allocation. [Research Report] RR-8007, INRIA. 2012, pp.23. <hal-00713693v2>

HAL Id: hal-00713693
https://hal.inria.fr/hal-00713693v2

Submitted on 2 Jul 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Polynomial Spilling Heuristic: Layered Allocation

Boubacar Diouf, Albert Cohen, Fabrice Rastello

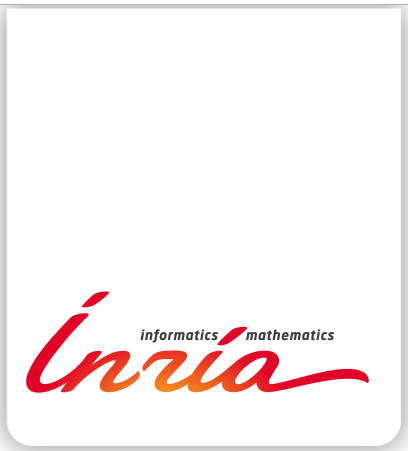
RESEARCH REPORT

N° 8007

July 2012

Project-Teams Parkas and Compsys

ISSN 0249-6399 ISRN INRIA/RR--8007--FR+ENG



A Polynomial Spilling Heuristic: Layered Allocation

Boubacar Diouf, Albert Cohen, Fabrice Rastello

Project-Teams Parkas and Compsys

Research Report n° 8007 — July 2012 — 23 pages

Abstract: Register allocation is one of the most important, and one of the oldest compiler optimizations. Its purpose is to map temporary variables to either machine registers or main memory locations and explicit load/store instructions. The latter option is referred to as spilling. This paper addresses the minimization of the spill code overhead, one of the difficult problems in register allocation. We devised a heuristic approach called *layered*. It is rooted in the recent advances in SSA-based register allocation. As opposed to the conventional incremental spilling approaches, our method incrementally *allocates* clusters of variables. We describe a new polynomial method, the layered-optimal allocator, and demonstrate its quasi-optimiality on standard benchmarks and on two architectures.

Key-words: compilers, algorithms, performance, optimizations, register allocation

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Une heuristique de spill polynomiale: l'allocation par couche

Résumé : L'allocation de registres est l'une des premiéres et des plus importantes optimisations effectuées par les compilateurs. Elle a pour but d'associer aux variables temporaires du programme des registres de la machine ou des locations mémoires et d'insèrer, dans le code, des instructions de load/store explicites, appelées vidage.

Dans ce papier, nous nous intéressons à la minimisation des latences mémoires dues au code de vidage, un des problèmes difficiles en allocation de registres. Nous proposons une approche heuristique d'allocation par couches. Ce travail se base sur les récentes avancées en allocation de registres sous SSA. Contrairement à l'approche conventionnelle de vidage incrémental, notre méthode alloue les variables de manière incrémentale par groupe. Nous comparons notre approche, appelée allocation-optimale par couche, aux methodes de l'état de l'art à une approche optimale et nous montrons l'allocation-optimale par couche est quasi-optimale sur des benchmarks standard et sur deux architectures différentes.

Mots-clés : compilateurs, algorithmes, les uns avec les autres optimisation, allocation de registres

1 Introduction

Register allocation is an important compiler optimization. Its goal is to map temporary variables in a program to either machine registers or memory locations. Register allocation is subdivided into two sub-problems: first, the *allocation* selects the set of variables that will reside in registers at each point of the program; then, the *assignment* or *coloring* picks a specific register where a variable will reside. Usually, all the variables of code cannot reside in registers. The variables not held in registers should reside in memory, these variables are called *spilled variables*. The *spilling* problem [13, 5] decides which variables should be stored in memory to make the assignment possible; it aims at minimizing the overhead of loads and stores. The *coalescing* [4] and *alienation* (when repairing is enabled [9]) problems aims at minimizing the overhead of moves between registers. Spilling and coalescing are correlated problems that are, in classical approaches, done in the same framework. Live-range splitting (i.e., adding register-to-register moves) to reduce register pressure is sometimes considered in such a framework [10], but it is very hard to control the interplay between spilling and splitting or coalescing.

Building on the properties of the static single assignment form (SSA), it is now possible to decouple the allocation from the assignment. Indeed, the interference graph of a program in SSA form is a chordal graph [16]. Since coloring a chordal graph is easy, it follows that the assignment problem is also easy. Finding a valid coloring whenever it exists can thus be solved optimally with a greedy, linear algorithm on chordal graphs, called *tree-scan* [9]. It follows the spirit of the linear-scan [18], but applied to the dominance tree instead [20]. Thus, performing register allocation under SSA has led to new approaches where the remaining difficult problems, spilling and coalescing, are treated separately. When spilling, MaxLive, the maximal number of variables simultaneously live at a program point, is used as a criterion to guarantee that the forthcoming assignment will be performed without any spill. If MaxLive is lower or equal to R, the number of available registers, then all the variables will be assigned without any spill. This *decoupled* approach is advocated by Fabri [12], Appel and George [2], and Hack [16].

Apart from allowing the design of more efficient coalescing heuristics [6], the main advantage of this decoupled approach concerns the spilling problem: checking if the register pressure, MaxLive, is low enough is much simpler than checking the colorability of a general graph. Because of this, existing graph based heuristics use node degree to guide the spilling decision. This can lead to spilling a variable because its corresponding node has a high degree, while it is not live at any point of high register pressure. In other words, spilling this variable is *useless* in helping the assignment problem anyhow. This point can be illustrated by Figure 1: variable a_2 has 6 neighbors of high weight (spilling each variable h_i is very costly), so spilling it looks like a good idea for the graph coloring. But in terms of register pressure, there are no more than three variables simultaneously live inside the loop: liveness set on the control flow graph, provides this information very naturally and a decoupled approach does not require spilling neither a_2 nor any h_i.

This observation has led several researchers to design *program-based* heuristics to lower register pressure, opposing the new decoupled approach to the "old" *graph-based* spilling heuristics. Remember that the decoupling approach eases the question of whether spilling a variable is useful or not, but did not make the optimization of allocation problem polynomial (finding a set of variables to be spilled of minimum cost). A simple heuristic consists in considering program points one after another, and when the register pressure at the current point is too high, *incrementally* spilling some variables to lower it. This incremental scheme needs a notion of profitability to choose which variable to spill among the set of all live variables at a given point. To this end, Belady's furthest first strategy works very well on an interval graph—a single basic block in SSA: the idea is to consider spilling the variable which live-range goes furthest to be the most prof-

itable. Consider our running example again where a_1 and a_2 have been coalesced into variable a. The generalization of the notion of "furthest use" to a general control flow graph would consider a to be more profitable than d. The furthest use is not the notion we want. Instead, profitability of spilling a variable should be related to the number of program points of high register pressure within the live-range of the variable. Here a would cover three high register pressure program points, while d would cover five. Going a little bit further, we understand that profitability is not just about coverage of high register pressure program points: spilling a variable is profitable because it avoids spilling some other variables, and because spilling it is less costly than spilling those other interfering variables. But what if we have to spill these interfering variables anyhow? As an example, spilling a avoids spilling d and e, but as it is very profitable to spill d (more than e), e should impact more than d the profitability of a, leading to an inductive definition of profitability... To break out of this loop, let us recall that under the SSA form, there exists a perfect mapping between maximal cliques and live variables at a given program point [16]. In other words, a maximal clique cover (which is polynomial for a chordal graph) allows to express the notion of register pressure, exactly for chordal graphs and reasonably accurately for non SSA graphs occurring in the real world [17]. On our running example, d would be part of two maximum cliques of size higher than three, while a would be in only one. This debunks the main motivation for not using a graph based approach.

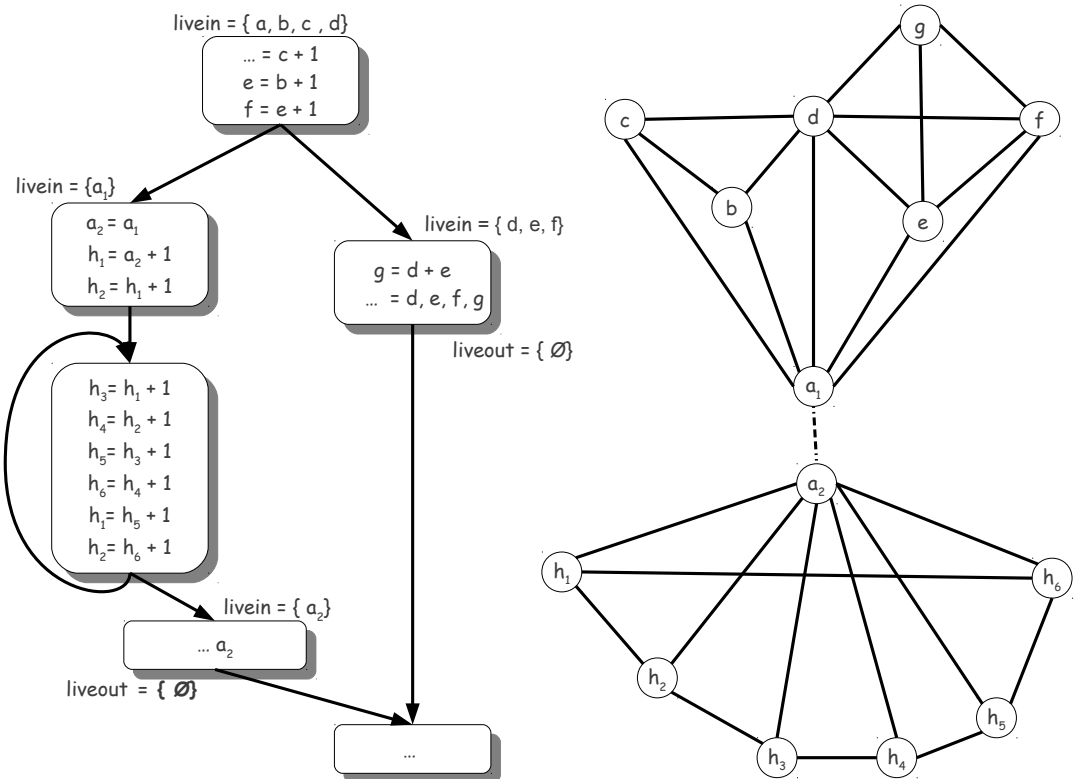


Figure 1: Program based, versus graph based spilling approaches on a program with three registers.

The goal of this paper is to propose a new graph-based allocation heuristic, based on the maxi-

mum clique cover to define the profitability of spilling variables. It exploits the pseudo-polynomial complexity in the number of registers of the allocation problem under SSA—as opposed to the symmetric, spilling problem which remains strongly NP-complete. More precisely, our approach emerges from two observations allowing for more global spilling decisions:

1. The pseudo-polynomial complexity of allocation in the number of registers [5] suggests a heuristic that solves (optimally) roughly R over *step* allocation problems on *step* registers each. The final allocation being the layered of the stepwise allocations, we call it the *layered-optimal heuristic*.

2. Stepwise optimality does not guarantee an overall optimal allocation, but we will show that it comes very close to optimal, even with $step = 1$. Intuition for this comes from recent work by Diouf et al. [11], observing that allocation decisions tend to be a monotonic function of the number of registers.

This approach is still incremental, but it allocates layers of variables instead of spilling one variable at a time. Thanks to the pseudo-polynomial property mentioned above, the choice of this set constituting a layer can be decided optimally in polynomial time.

To make a long story short, this paper addresses the spill-everywhere problem in a decoupled context. We introduce *layered allocation* a new strategy that incrementally allocates variables instead of incrementally spilling variables. We evaluate our approach in the context of decoupled register allocation.

The outline of the paper is as follows. Section 2 presents the rationale for our new approach in detail. Section 3 surveys the important concepts and results upon which our approach is built. Section 4 presents our layered-optimal allocator for SSA programs. Section 5 adapts this scheme into a non-optimal heuristic for general, non-chordal interference graphs. Section 6 evaluates the algorithm and compares it with state of the art methods. Section 7 discusses related work and Section 8 concludes the paper.

2 A Graph-Based, Incremental Allocation Approach

Our approach is motivated by the facts that register lowering is pseudo-polynomial in the number of registers and stepwise allocation is quasi-optimal. We explain here in details these two observations, but before that, we first explain why we think that the spill everywhere problem is relevant.

2.1 Why Spill Everywhere?

The spilling problem can be considered at different granularity levels: the highest, so called *spill everywhere*, corresponds to considering the live range of each variable entirely. A spilled variable will then lead to a store after the definition and a load before each use. Of course, in practice, if the variable can stay in a register between two consecutive uses, a load is saved. The finest granularity, so called load-store optimization, corresponds to optimize each load and store separately. The latter, also known as paging with write back, is NP-complete [13] on a basic block, even under SSA form. The spill-everywhere problem is much simpler, applicable to just-in-time compilation, and many instances are polynomial under SSA form [5]. The algorithms we propose can be applied to both spill everywhere and load-store optimization problems. We focus here on the former for its simplicity, because our past experience summarized in the 4 following points tends to confirm the practical effectiveness of the spill everywhere problem:

1. The complexity of the load-store optimization problem comes from the asymmetry between loads and stores. Also, most SSA variables have only one or two uses in practice, and the cost of the store favors spilling the entire live range instead of two sub-ranges of different variables.

2. The queuing mechanism present in most architectures behave like a small, extremely fast cache. But it is highly sensitive to the number of simultaneously spilled variables.

3. In the other extreme situation where stores have no cost, a variable can be considered to be either in memory or in register but not in both. Such a formulation [2] is strictly equivalent to a spill everywhere formulation where live ranges are split at every use.

4. Last, a solution to the spill-everywhere problem gives to a load-store optimization problem the global view lengthily discussed so-far that existing heuristics lack. In other words a spill-everywhere solution can play the role of an oracle.

2.2 Allocation Instead Of Spilling

After giving the reasons that support our work on the spill everywhere problem, let us stress the difference we want to make here between spilling and allocation. Spilling aims at finding which variable to evict from registers while allocation aims at finding which variable to keep in registers. Of course, one is the dual of the other, so conceptually spilling and allocation are the same. Now suppose you have a set of variables and you want to evict (spill) a minimum amount of them such that MaxLive is lowered by just one. As shown in [5] this problem is NP-complete even for the simplest SSA program instance. On the other-hand, consider you have already a set of allocated variables and you aim at allocating a maximum number of additional ones such that at every program point the register pressure Live is increased by at most 1. Then as outlined before, this problem is, under SSA, polynomial with a complexity of $\mathcal{O}(\Omega n)$. Ω being the maximum simultaneously live variables that remains to be allocated; n being the size of the program. Hence, in a way allocation is simpler than spilling. Our approach pushes this distinction further: Conceptually, every variable is initially in memory, and we evaluate the *gain* of allocating a given one instead of considering every variables to be initially in a virtually unbounded register file and evaluate the *cost* of evicting it. As we will see in this paper, this allows to be much more accurate concerning the modeling of gain/cost that accounts for ABI and register constraints.

2.3 Stepwise Allocation Is Close To Optimal

In a recent paper, Diouf et al. [11] studied the question to know whether or not the variables spilled on an optimal allocation with R registers are included in the set of variables spilled on an optimal allocation with R -1 registers ($R > 0$). Conceptually, this is equivalent to telling that the variables allocated on an optimal allocation with $R - 1$ registers are included in the set of variables allocated when R registers are available. The answer to the question is no and this is illustrated in Figure 2. In this figure, we give the graph version of the example used by Diouf et al. [11]. The cost of each variable is represented by the number close to its corresponding node. Dashed black circles correspond to spilled variables. Figure 2(a) depicts the optimal allocation performed when $R = 1$ and Figure 2(b) shows the optimal allocation performed when $R = 2$. When $R = 1$, to perform the allocation of lowest cost, we need to spill b and d, which form the optimal spill set. When $R = 2$, we need to spill c, which is the optimal spill set. We clearly see that the optimal spill set when $R = 2$ is not included in the optimal spill set when $R = 1$.

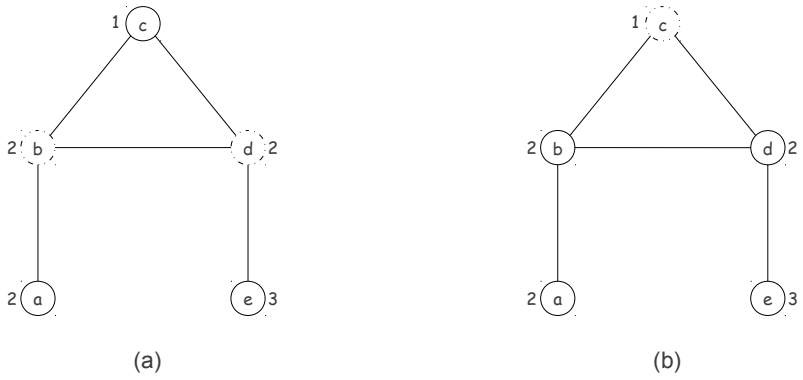


Figure 2: Counter example to spill set inclusion.

Even if, theoretically, the answer to the question of spill set inclusion is no, Diouf et al. experimentally validated that when varying the number of registers from R_{min}, the minimum number of registers to enable code generation, to the number of registers allowing to allocate all the variables, the inclusion property holds for 99.83% of the SPEC JVM98's methods. This experimental evaluation has been done with JikesRVM [1], the research virtual machine of IBM. This also proves, empirically, that the stepwise allocation is close to optimal.

3 Baseground

We now summarizes some definitions and results on graphs and chordal graphs upon which our approach is based.

In the rest of this paper, we assume that an estimated spill cost has been computed for each variable. A spill cost represents the access frequency of a variable, it is high when the variable is frequently accessed and low when it is not. We denote R the number of available registers.

Programs are usually represented as graphs, within graph coloring frameworks, and live sets within linear scan frameworks. Thus the spilling problem is naturally solved over these two representations. Our approach is compatible for both representations, but in the rest of this section we will focus on the graph representation.

3.1 Graphs and Weighted graphs

A *graph* $G = (V, E)$ consists of two sets, V the set of vertices or nodes, and E the set of edges. Every edge (v_1, v_2) of E has two end points $v_1 \in V$ and $v_2 \in V$. We say that v_1 and v_2 are *adjacent(s)* or are *neighbor(s)* if $(v_1, v_2) \in E$. The number of neighbors of a vertex v is called the *degree* of v. Here, We only consider *undirected* graphs, i.e., we do not make difference between the edges (v_1, v_2) and (v_2, v_1). Figure 3(a) shows an arbitrary graph.

A sequence of vertices $[v_0, v_1, v_2, \ldots, v_l, v_0]$ is called a *cycle* of length $l + 1$ if $(v_{i-1}, v_i) \in E$ for $i = 1, 2, l$ and $(v_l, v_0) \in E$.

A subset $A \subseteq V$ is called a *clique* of G if every two distinct vertices of A are adjacent. A clique A is *maximal* if it is not properly contained in any other clique of G. A clique is *maximum* if there is no clique of G of larger cardinality. A vertex v of a graph G is *simplicial* if its neighbors

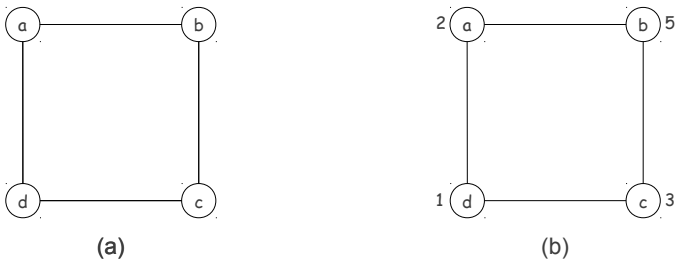


Figure 3: (a) An arbitrary graph. (b) A weighted graph.

form a clique in G.

In contrast to a clique, a *stable set* or an *independent set* is a subset $S \subseteq V$ that does not contain two vertices that are adjacent.

Assuming each vertex v of $G = (V, E)$ is associated with a non-negative number $w(v)$, the weight of a subset $S \subset V$ is expressed as:

$$w(S) = \sum_{v \in S} w(v)$$

The graph G associated with the function w is called a weighted graph and denoted G_w. Figure 3(b) shows a weighted graph based on the arbitrary graph presented in Figure 3(a). Each vertex has its weight close to it. For instance the vertex a has a weight of 2.

The maximum weighted stable set is the stable set of maximal weight.

From a graph representation of a program, if we associate to each vertex a weight corresponding to its cost (it is assumed that a cost has been computed for each variable), the spilling problem becomes equivalent to the problem of choosing the set of vertices of minimal weight to remove from a weighted graph to make a coloring/assignment possible. This problem is much more complicated on arbitrary graphs, since the coloring problem is already NP-complete on these category of graphs. This coloring problem becomes easy on chordal graphs which are discussed below.

3.2 Chordal Graphs

The static single assignment (SSA) form is an intermediate representation with very interesting properties. A code is in SSA form when every scalar variable has only one textual definition in the program code. Most compilers use a particular SSA form, the strict SSA form, with the additional so-called dominance property: given a use of a variable, the definition occurs before any uses on any path going from the beginning of the program (the root) to a use. One of the useful properties of such a form is that the dominance graph is a tree and the live ranges of the variables (delimited by the definition and the uses of a variable) can be viewed as subtrees of this dominance tree. The intersection graph of these subtrees of the dominance tree represents the interference graph. An important result of graph theory states that the intersection graph of a family of subtrees of a tree is a chordal graph [15]. It follows that the interference graph of a program in SSA form is a chordal graph.

A graph G is *chordal*, *triangulated* or *rigid-circuit* if every cycle of length four or more has a chord, a chord being an edge joining two vertices of the cycle, that are not consecutive. The

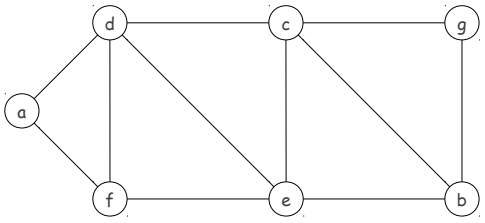


Figure 4: An example of chordal graph.

graph given in Figure 3(a) shows a non-chordal graph and Figure 4 shows a chordal graph, for instance the cycle $[c, d, f, e, c]$ has a chord which is (d, e).

Algorithm 1 MaximumWeightedStableSet

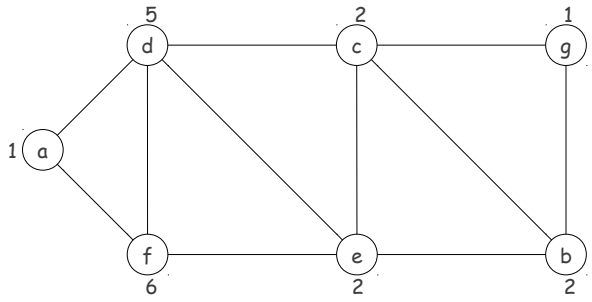
Require: σ: a perfect elimination order
Require: w: a map associating to each vertex its weight
Require: *adj*: a map associating to each vertex the list of its neighbors
Var: *w'*: a map associating to each vertex its current weight during computation
Var: n: the number of vertices
Var: *marked_red*: a (last in first out) list keeping track of vertices marked red
Var: *marked_blue*: a list keeping track of vertices marked blue

```
1:  for i = 1 → n do
2:     v ← σ(i)
3:     w'(v) ← w(v)
4:  end for
5:  for i = 1 → n do
6:     v ← σ(i)
7:     if w' > 0 then
8:        add v to marked_red
9:        for all u ∈ adj(v) do
10:          w'(u) ← w'(u) − w'(v)
11:          if w'(u) < 0 then
12:             w'(u) ← 0
13:          end if
14:       end for
15:       w'(v) ← 0
16:    end if
17: end for
18: while marked_red ≠ ⊥ do
19:    v ← the first element of marked_red
20:    Remove v from marked_red
21:    Add v to marked_blue
22:    remove all the vertices of adj(v) from marked_red
23: end while
24: return marked_blue
```

An interesting property that we are going to use below in the paper is that it is easy to compute the maximum weighted stable set of a chordal graph [14] and this with a complexity of $\mathcal{O}(|E|+|V|)$.

Before explaining the Frank's algorithm which computes the maximum stable set of a weighted graph, we need to explain the notion of perfect elimination order. An ordering $v_1, v_2, \ldots, v_n$ of the vertices of a graph G is a *perfect elimination order* (PEO) if each v_i is a simplicial vertex in $G_{\{v_i, v_{i+1}, \ldots, v_n\}}$, the graph remaining from G when all the vertices preceding v_i in the ordering have been removed. It has been proven in graph theory that a graph is chordal if and only if it has a perfect elimination order [15]. For instance $[a, f, d, e, b, g, c]$ is a PEO of the chordal graph given in Figure 4.

Algorithm 1 computes the maximum weighted stable of a weighted graph G_w. It receives σ a perfect elimination order of the weighted chordal graph, *adj* a map that associates to each vertex the list of its neighbors and w the weight function that associates to each vertex of G_w its weight. Algorithm 1 goes through the list of vertices according to the order of σ. At the step i, it tests if the current weight of the vertex v that occupies the $i-th$ position in σ is positive. If it is negative Algorithm 1 goes to the next step. Otherwise, v is marked red and each neighbor u of v has its current weight $w'(u)$ reduced by $w'(v)$. Any weight that becomes negative is altered to 0 and $w'(v)$ is set to 0. At the end of this process, the vertices marked red are visited in the reverse order of their insertion in the list. A vertex v is marked blue if it is not a neighbor of all the vertices previously marked blue. Finally, Algorithm 1 returns the set of vertices marked blue, that is stable set of maximum weight.



(a)

iteration	a	f	d	e	b	g	c	red vertices
-	1	6	5	2	2	1	2	Ø
1	0	5	4	2	2	1	2	a
2		0	0	0	2	1	2	f, a
5					0	0	0	b,f,a

(b)

iteration	red vertices	blue vertices
-	b,f,a	Ø
1	f,a	b
2	Ø	b, f

(c)

Figure 5: Looking for the maximum weighted stable set with Algorithm 1.

Figure 5(b) and Figure 5(c) depicts the general steps of Algorithm 1 when applied to the

graph given in Figure 5(ca. Figure 5(b) shows how the set of vertices marked red is constructed. The column *iteration* presents the iterations of the second for-loop of Algorithm 1 that modifies the set of vertices marked red. The second column keeps track of the values of w'. The vertices are ordered according to the perfect elimination order. The last column shows how the set of marked red evolves. The first row shows, before the beginning of the loop, the values of w' for each vertex and the set of vertices marked red which is empty. At the first iteration, the weight of a is 1, thus a is marked red and the weights of its neighbors d and f are decreased by 1. The weight of a is then set to 0, which is underdrew. At the second iteration, the weight of f is 5. Thus, f is marked red and the weights of its neighbors a, d and e are decreased by 5. The weight of f is then set to 0, which is underdrew. Finally, we obtain the set of vertices marked red which are composed of b, f, a.

Figure 5(c) explains how from the vertices marked red we compute the set of vertices marked blue. This is performed with the while-loop of Algorithm 1. At the first iteration, the vertex b is chosen and is inserted in the set of vertices marked blue. At the second iteration the vertex f is chosen and inserted in the set of vertices marked blue. The vertex a is adjacent to f and cannot be added to the set of vertices marked red. Thus, a is removed from the list of vertices marked red. We then end up with a set of vertices marked blue composed of f and b of weight 8.

4 Layered-Optimal Register Allocation

We will focus here on the spilling problem for SSA programs. In the next section, we will give an extension of our approach which works on general graphs.

Based on the two observations explained on Section 2, we present here our solution which solves the spill minimization problem for R registers by layered optimal solutions to simpler problems on few registers. Each of this simpler problem is considered to have *step*, which is a small number lower or equal to R, available registers. Stepwise optimality does not guarantee an overall optimal allocation, but we will show that it comes very close to optimal, even with $step = 1$.

Algorithm 2 LayeredOptimalAllocation

Var: *candidates*: the list of vertices that are candidate to an allocation
Var: *allocated_list*: the list of so far allocated variables
1: $count \leftarrow 0$
2: **while** $candidates \neq \perp \wedge count < R$ **do**
3: $result \leftarrow$ OptimalAllocation(*candidates*)
4: add every vertex of *result* to *allocated_list*
5: remove every vertex of *result* from *candidates*
6: $count \leftarrow count + 1$
7: **end while**
8: **return** *allocated_list*

Algorithm 2 implements the layered-optimal heuristic. It takes as input *candidates*, the list of variables that are candidates to register allocation. It then returns as result *allocated_list*, the list of variables that have been allocated with R registers. Algorithm 2 calls OptimalAllocation which returns the *optimal allocation set* minimizing the spill cost among the variables that have not yet been allocated (currently in *candidates*). This set is added to *allocated_list* and removed from *candidates*. In its last step, Algorithm 2 finds the set of variables that minimizes the spill cost among the variables remaining in *candidates*.

The function OptimalAllocation used in Algorithm 2 solves the allocation problem on a subset of the candidates variables when a unique register is available. On a chordal interference graph, this problem is equivalent to the problem of the maximum weighted stable set. Thus, the OptimalAllocation can be implemented with Algorithm 1. When assuming that $step \geq 2$, OptimalAllocation can be implemented through dynamic programming [5].

In the following, we restrict ourselves to a step of one. The complexity of the layered-optimal allocator is $\mathcal{O}(R(|V| + |E|))$.

Algorithm 2 is a solid basis for an incremental allocation, but we have found two ways to improve it: biasing the cost of the variables, and iterating further on the set of allocated variables until we reach a point where we could not allocate more variables.

4.1 Biasing the weights

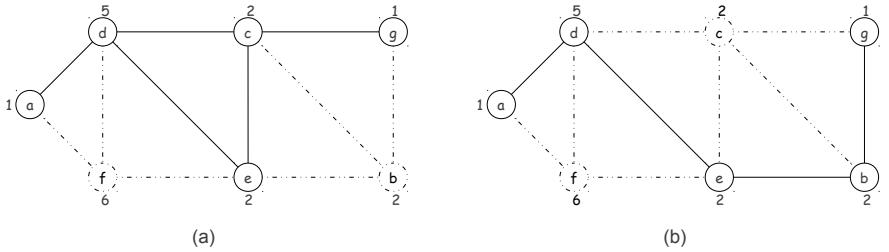


Figure 6: Example showing the benefit of biasing the weight.

Before we explain how we bias the costs/weights of variables/weights, let us first have a look on Figure 6. We assume we have two registers, the step is set to one, and we are looking for the set of variables to allocate for the weighted graph given in Figure 5(a). We call this graph G_w. When called on G_w, Algorithm 2 will first look for the maximum weighted stable set on the graph and consider the variables within this set as allocated. It will then look for the maximum weighted stable set on the graph remaining when the allocated variables are removed. G_w has two maximum weighted stable sets with a weight of 8. The first one shown in Figure 6(a) is composed of vertices b and f, in dashed lines. The second one composed of c and f, also in dashed lines, is shown in Figure 6(b). If b and f are chosen, at the next step, Algorithm 2 will look for the maximum weighted stable set on the graph remaining when b and f are removed from G_w. This graph is represented by the black nodes and edges in Figure 6(a). The maximum weighted stable set for this graph is composed of d and g and has a cost of 6. This leads to spilling variables a, c and e with a spill cost of 4. In contrast, if we choose c and f, at the next step the maximum weighted stable of the graph, shown in black nodes and edges in Figure 6(b), will be composed of b and d with a cost of 7. This lead to a spill cost of 3.

This example shows that the choice among different maximum weighted stable sets has an impact on the next iterations of Algorithm 2. Arbitrarily Choosing a maximum weighted stable set can deteriorate the global register allocation. Our intuition to ameliorate the choice of the maximum weighted stable set is that: it is almost always better to choose the maximum weighted stable set that removes the most interferences in the graph on non-allocated variables. Our approach to achieve this is to bias the cost/weight of variables/vertices with the number of neighbors of a vertex and we define, to this purpose, the new weight function w' as:

$$w'(v) = w(v) \times |V| + |adj(v)|$$

where $|V|$ is the number of vertices of the graph and $adj(v)$ is the number of neighbors of the vertex v.

For two vertices u and v, the two following properties will always be verified with the new weight function:

$$\begin{cases} \text{if } w(u) < w(v) \text{ then } w'(u) < w'(v) \\ \text{if } w(u) = w(v) \text{ then } w'(u) \leq w'(v) \text{ if } adj(u) \leq adj(v) \end{cases}$$

4.2 Iterating To Fixed Point

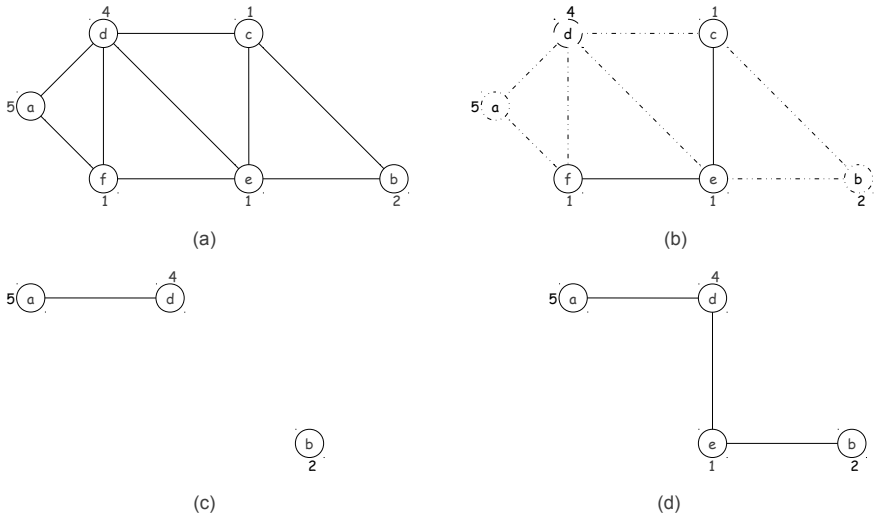


Figure 7: Example showing the benefit of iterating until a fixed point.

We introduce here the second improvement we want to perform on Algorithm 2. We assume we have two registers, the step is set to one, and we are looking for the set of variables to allocate for the weighted graph given in Figure 7(a). This graph is called G_w and has four maximal cliques which are: $\{a, d, f\}$, $\{b, c, e\}$, $\{c, d, e\}$, and $\{d, e, f\}$. Figure 7(b) shows the set of allocated variables returned by Algorithm 2 which are composed of vertices a, b and d, shown in dashed lines. Algorithm 2 ends up with this set of allocated vertices whether or not the weights are biased. Figure 7(c) shows the graph formed of allocated variables found by Algorithm 2. Let us recall that a coloring with R colors is possible on a chordal graph if the maximum clique of the graph does not have more than R vertices. If we focus on the vertex f, we notice that in the graph G_w, f belongs to a maximal clique composed of a, d and f, which have 2 (R)vertices already allocated. The vertex f cannot be added to the graph of allocated vertices shown in Figure 7(c) without adding a clique of size 3 and thus making a coloring with 2 colors impossible. Unlike f, the vertices c and e are not contained in a maximal clique that has 2 vertices already allocated. It follows that either c or e can be added to the graph of allocated vertices shown in Figure 7(c) without making a coloring, with two colors, of the graph impossible since the size of the maximum clique of the resulting graph, shown in Figure 7(d), will be 2.

Algorithm 3 FIXEDPOINTLAYERED (FPL)

Require: *candidates*: the list of vertices that are candidate to an allocation
Var: *allocated_list*: the list of so far allocated variables
Var: *allowed_cliques*: the list of cliques that do not have more than R allocated vertices, it is initialized to the list of maximal cliques
Var: *allocated_Per_clique*: a map associating to each maximal clique the number of its allocated vertices

```
1:  count ← 0
2:  while candidates ≠ ⊥ ∧ count < R do
3:     result ← OPTIMALALLOCATION(candidates)
4:     add every vertex of result to allocated_list
5:     remove every vertex of result from candidates
6:     count ← count + 1
7:  end while
8:  UPDATE(candidates, allocated_list, allowed_cliques, allocated_Per_clique)
9:  while candidates ≠ ⊥ do
10:    result ← OPTIMALALLOCATION(candidates))
11:    remove every vertex of result from candidates
12:    UPDATE(candidates, result, allowed_cliques, allocated_Per_clique)
13: end while
14: return allocated_list
```

Algorithm 4 UPDATE

Require: *candidates*: the list of vertices that are candidate to an allocation
Require: *allocated_list*: a list of allocated variables
Var: *allowed_cliques*: the list of cliques that do not have more than R allocated vertices
Var: *allocated_Per_clique*: a map associating to each maximal clique the number of its allocated vertices

```
1:  for all v ∈ allocated_list do
2:    for all clique ∈ allowed_cliques do
3:      if v ∈ clique then
4:        increment allocated_Per_clique(clique) by one
5:        if allocated_Per_clique(clique) ≥ R then
6:          remove all the vertices of clique from candidates
7:          mark clique as to be removed at the end of the loop
8:        end if
9:      end if
10:   end for
11: end for
```

The example given in Figure 7 shows that we could miss interesting allocations when naively using the Algorithm 2. Algorithm 3 first performs the layered allocation iterating at most R times (lines 1 to 7). It then calls Algorithm 4 which increments, for each freshly allocated vertex, the number of allocated vertices of each clique which contains it. If a clique has R of its vertices allocated, all the vertices of this clique are removed from *candidates* and this clique is removed from the list of *allowed_cliques*, which are the cliques that can have one of their non-allocated vertices allocated at next rounds. After Algorithm 4 finishes, Algorithm 3 calls

the function OptimalAllocation and Algorithm 4 iteratively until it reaches a fix point, that is, an allocation that cannot be improved by subsequent calls of Algorithm 3.

4.3 Spilled variables

When a variable is spilled, it does not completely disappear from the interference graph. It is replaced by a set of short-lived variables which must be taken into account. For instance on RISC architectures, memory can only be accessed through load and store instructions. For example, before using a variable v spilled at address a, the value of v must be loaded from address a into a register. The extra instructions inserted to reload spilled variables form the *spill code*.

Some approaches—like the JikesRVM implementation of the linear scan—spill locally an allocated variable when there is no free register to assign a reloaded variable. On CISC architectures like the x86, we also can take advantage of complex addressing modes to get operands directly from memory (at most one such operand on x86). On the other hand, graph coloring heuristics iteratively rebuild interferences after spill. Symmetrically, we can iteratively update the interferences after allocation.

5 Layered-Heuristic Allocator

Although the spill minimization problem is only pseudo-polynomial on SSA programs, the method also applies to general programs. The layered approach remains applicable, but the Frank's algorithm is not applicable as the graphs are not chordal.

The layered-heuristic algorithm clusters the nodes of an interference graph as stables—or independent sets—using a greedy heuristic. It clusters the variables according to their interference and spill costs, then allocates registers into layered cluster-based allocations.

We cluster variables that do not interfere and that corresponds to a stable set in the interference graph. Since we cannot compute the maximum weighted stable set in an arbitrary graph in a polynomial time, we approximate it. The performance of the allocator will depend on the quality of the approximated weighted stable set of maximum weight, which is called a *cluster*. The clusters are computed incrementally, that is, we first compute a cluster and then another cluster which does not contain any variable of the first cluster, and so on, until we put all the variables into clusters. To approximate a cluster, we incrementally merge high-weights nodes which do not interfere with the variables already present in the stable set.

The clustering is performed by Algorithm 5 which transforms *candidates*, a list of vertices of a graph sorted by decreasing weight, into *cluster_list*, a list of clusters. It constructs a new cluster at each iteration of the outer while-loop. In order to compute *cluster*, the new cluster, all the variables still in *candidates* are added to *potentials*. The list *potentials* keeps tracks, at each round of the inner while-loop, the vertices that do not interfere with the vertices already into *cluster*. Every time a vertex v is added to *cluster*, all the neighbors of v are removed from *potentials*. At the end of the inner while-loop, the computed *cluster* is added to the *cluster_list*, and the next round of the outer while-loop starts. Finally, Algorithm 5 ends when every variable is in a cluster.

After the clusters have been computed, Algorithm 6 decides which clusters should be allocated to registers. The R clusters that maximize the sum of their weights are allocated.

The complexity of the layered-heuristic allocation is $\mathcal{O}(R \times (|V| + |E|))$. Indeed the algorithm iterates at most R times for R registers, the clustering step visits every neighbor of a node only once.

Algorithm 5 ClusterVertices

Require: *candidates*: a list of vertices, ordered by decreasing weight, that are candidate to an allocation
Require: *adj*: a map associating to each vertex the list of its neighbors
Var: *cluster_list*: a list of clusters

```
1:  while candidates ≠ ∅ do
2:     cluster ← ⊥
3:     // add all the vertices of candidates to potentials
4:     potentials ← candidates
5:     while potentials ≠ ∅ do
6:        remove from potentials its first vertex, called v
7:        add v to cluster
8:        remove all the vertices of adj(v) from potentials
9:     end while
10:    add cluster to cluster_list
11:    remove all the vertices of cluster from candidates
12: end while
13: return cluster_list
```

Algorithm 6 AllocateVertices

Require: *candidates*: a list of vertices that are candidate to an allocation
Require: R: the number of machine registers

```
cluster_list ← ClusterVertices(var_list)
sort cluster_list by decreasing cost
if sizeof(cluster_list) > R then
   remove the last (size − R) clusters from cluster_list
end if
spill each variable not in cluster_list
```

6 Experimental Evaluation

Our approach is very well suited to SSA programs, but we show that, it also yields excellent results on arbitrary interference graphs from non-SSA programs.

6.1 Chordal Graphs: SSA Programs

6.1.1 Methodology

We evaluated our approach on chordal interference graphs resulting from programs compiled with the Open64 compiler for the ST231 VLIW processor and for the ARM Cortex A8 (ARMv7). For the former, we generated the interference graphs for the *SPEC CPU 2000int*, the *lao-kernels* (an internal suite from STMicroelectronics) and the *eembc* benchmarks. We only used the *lao-kernels* for the ARMv7 processor.

For each of the considered benchmarks, we computed the spill costs based on the basic blocks's frequency and on the number of accesses to the variables within the basic blocks. We studied the impact of the register count, ranging from 1 to 32. For each instance of the register allocation problem and for each configuration, we compared the following algorithms:

GC The Chaitin-Briggs, optimistic graph coloring algorithm.

Optimal An optimal ILP-based allocator.

NL The layered allocation method implemented without the two improvements presented in Section 4.

FPL The layered allocation method with the fixed point improvement that incrementally allocates variables until no variable could be allocated any more, with the given register count.

BL The layered allocation method that biases the spilling cost of nodes in order to choose between two stable sets of same (un-biased) cost the one that has more interferences.

BFPL The layered allocation method which biases the cost and iterates until a fixed point in the allocation.

6.1.2 Results and discussion

The results obtained from the evaluation of chordal graphs generated from SPEC CPU 2000int, lao-kernels and eembc benchmarks are very similar.

Figure 8: Allocation cost for the SPEC CPU 2000int benchmark suite on ST231.

Figure 9: Allocation cost for the EEMBC benchmark suite on ST231.

Figure 8 presents the average of the allocation cost of all the application of the SPEC CPU 2000int. For the sake of exposition, we reported here the results for configuration with a register count of 1, 2, 4, 8, 16 and 32 registers. For all the configuration, BL, FPL, BFPL are close to optimal on average and are better than GC. On configuration with register counts up to 8, BL is

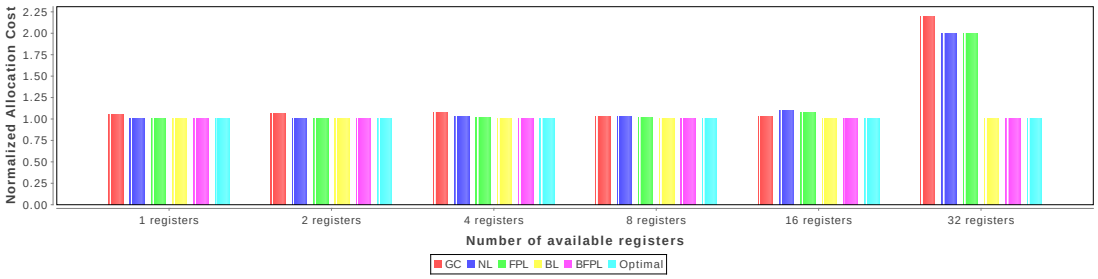


Figure 10: Allocation cost for the LAO-KERNELS benchmark suite on ARMv7.

also quasi-optimal, but for configuration with 16 and 32 registers, we notice a performance degradation. This is reinforced by Figure 9 and on Figure 10 we also notice a performance degradation, when the register count is 32, of the FPL approach; it suggests that the biased improvement is very helpful on the lao-kernels benchmark suite, which is made of small benchmarks and thus can be more impacted by a bad allocation choice.

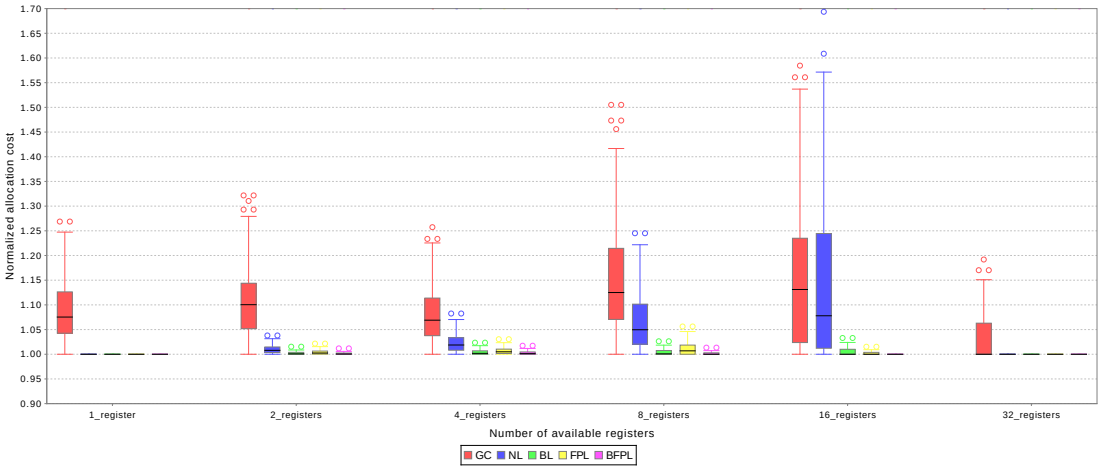


Figure 11: Distribution of the allocation costs over individual programs of the SPEC CPU 2000int benchmark suite on ST231.

Figure 11 studies how the allocation results vary across individual interference graph for all the benchmark programs in the SPEC CPU 2000int suite. Each allocation result is normalized to the optimal allocation for the specific benchmark. This figure depicts the distribution of these normalized allocation costs. GC, and to a lesser extent NL, show a high variability. This indicates that some benchmarks yield poor allocations for these allocators. On the contrary, BL, FPL and BFPL are consistently successful at computing close-to-optimal allocations. This is confirmed by Figures 12 and 13 on the other benchmark suites. Notice a slight variability for FPL and registers on the lao-kernels targetting the ARMv7 (Figure 13).

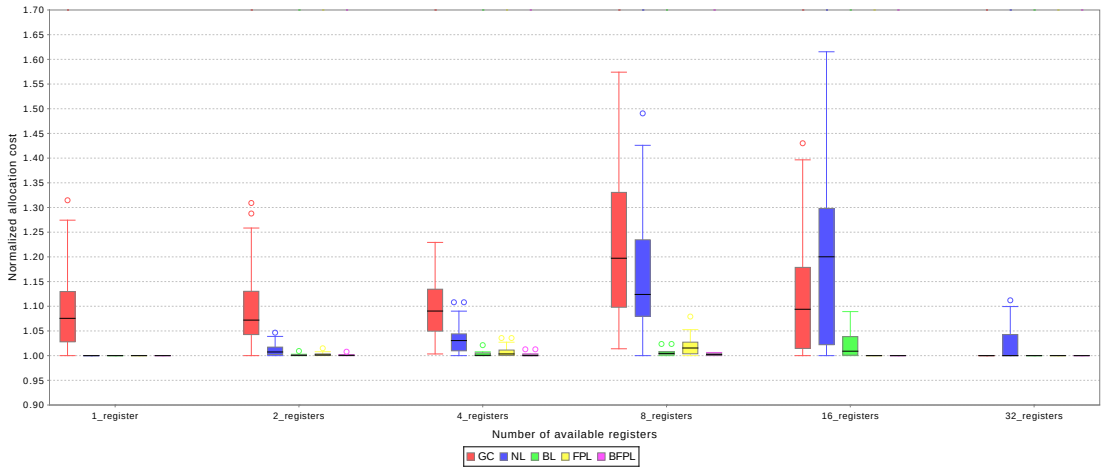


Figure 12: Distribution of the allocation costs over individual programs of the EEMBC benchmark suite on ST231.

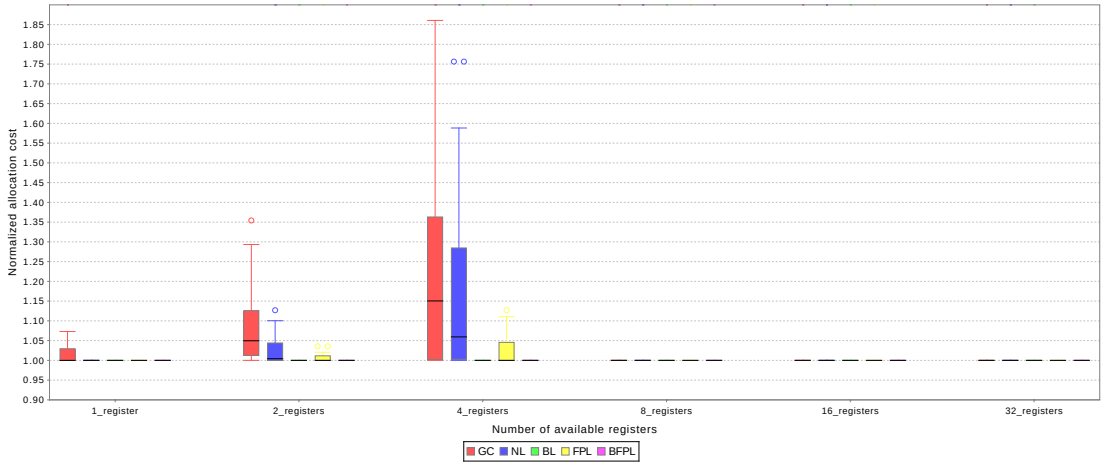


Figure 13: Distribution of the allocation costs over individual programs of the lao-kernels benchmark suite on ARMv7.

6.2 Extension To Non-Chordal Graphs

We evaluate our approach on general, non-SSA progams, studying the *SPEC JVM 98* benchmark suite (a benchmark set to measure the performance of Java virtual machines). We use the JikesRVM just-in-time compiler; its intermediate representation is not in SSA, and the interference graphs are not chordal in general.

We considered different configurations of register count going from 2 to 16. For each instance of the register allocation problem and for each configuration, we compared the following algorithms:

LS The original linear scan algorithm as implemented in JikesRVM.

BLS A variant of the linear scan relying on Belady's furthest-first strategy to make spilling decisions if their costs are close enough according to a chosen threshold.

GC The Chaitin-Briggs, optimistic graph coloring algorithm.

Optimal The globally optimal allocation implementing an ILP model proposed by Diouf et al. [11].

LH Our layered-heuristic method.

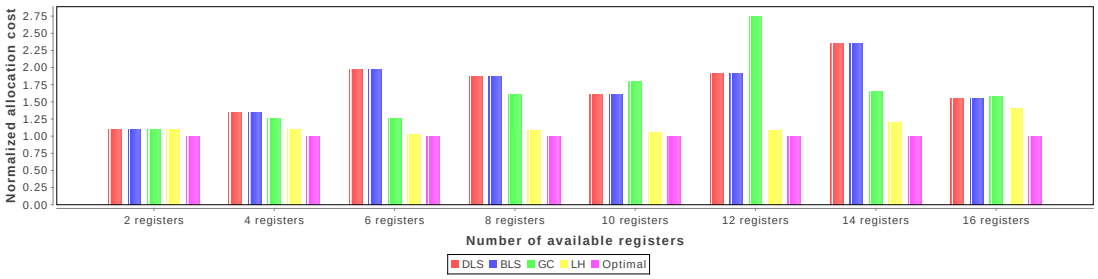


Figure 14: Layered-heuristic allocator compared to other algorithms for different register counts.

Figure 14 shows the allocation costs for all SPEC JVM together, normalized over the cost of the optimal allocation's cost. Configurations with different register counts going from 2 to 16 registers. For almost all the register counts, the layered-heuristic allocator is close to optimal, except for the configurations with 14 and 16 registers. This can be explained by the accumulation of approximations in the incremental construction of maximal weighted stables, a consequence of the non-chordality of the interference graphs.

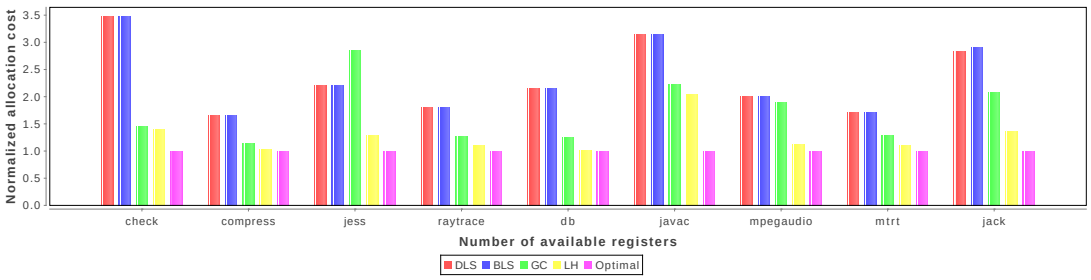


Figure 15: Layered-heuristic compared to other allocators when the register count is 6.

Figure 15 reports for each individual benchmark the normalized allocation costs when we have a register count of 6 registers. We see that here again, the layered-heuristic allocator performs close to optimal allocations, and outperforms all the other allocation heuristics. For `check`, `jess`, `javac`, and `jack`, the overhead can reach 60% of the optimal, but the cost is still better than the conventional heuristics.

7 Related Work

Register allocation algorithms often rely on spilling algorithms to perform spill minimization.

In static compilation the dominant approach to register allocation is the graph coloring in which the spilling and coloring (assignment) algorithms are interleaved. During the *simplify phase*, whenever all the remaining nodes have at least R degrees, a node needs to be marked as spilled or pushed onto the stack (optimistic coloring) and removed from the graph. A natural intuition is to choose a node that has a low spilling cost and which interferes a lot. Many of the graph coloring variant are based on this intuition and use the quantity $cost(v)/deg(v)$ to choose the variables to spill [8]. Thus, the spilling algorithm uses a global information over the whole program that combines the interference degree and the spilling cost.

In the context of just-in-time (JIT) compilation, register allocation time is part of the global execution time and (quasi-)linear complexity remains a driving force in the design of optimization algorithms. Moreover, when embedded systems are addressed, the limited memory resources is also an important issue. The linear scan which is one of the most used register allocation algorithm on JIT compilers has a worst case complexity of $\mathcal{O}(n \times R)$, where n is the number of variables in the program and R is the number of available registers on the target architecture. The original spilling heuristic used in linear scan [18] is based on the Belady's furthest first algorithm [3]. This algorithm relies on *local information* to perform spilling: "At a point p where registers are not enough to hold all the live variables, spill the variables whose live ranges go farther in the future". Recent versions of linear scan use more elaborate algorithms based on variables' spill cost estimation and sharing some of the global spilling decisions of graph coloring [21].

The idea of improving the spill minimization in a decoupled approach, where the allocation is decoupled from the assignment, has been explored by Proebsting and Fischer [19], and by Braun and Hack [7]. Braun and Hack generalized the Belady's furthest first algorithm — which works very well on straight-line code — to control-flow graphs. Their approach, while being applicable as a pre-spill phase in any compiler, is more adapted to SSA-based register allocation. They reported a reduction in the number of reload instructions by 54.5% compared to the linear scan and by 58.2% compared to the graph coloring. An other approach approach by Pereira and Palsberg rely on maximal cliques to drive spilling decisions with similar goals on chordal graphs, and generalizability to general graphs [17]. Like the latter approaches, layered allocation is fast and can be used in a non-decoupled context for general programs, in a decoupled context for SSA programs, and as a pre-spill phase in any compiler. Unlike Braun and Hack, we experimentally show how that our layered-optimal algorithm performs close-to-optimal allocations.

8 Conclusion

Combining key observations in SSA-based, decoupled register allocation, we designed a new, polynomial approach to the spill-cost minimization problem: layered allocation. Our method contrasts with decades of work on register allocation by incrementally allocating clusters of variables to registers, while conventional heuristics incrementally spill variables. The criterion to form these clusters, rooted in the maximal clique problem (polynomial on chordal graphs), is also original. Our algorithm produces allocations that are very close to optimal on SSA programs, outperforming higher complexity heuristics such as the graph coloring methods. We also adapt our method to design an allocation heuristic for general, non-SSA programs.

These fundamental results pave the way to a simpler and very effective register allocation framework. Several steps remain to be taken to integrate it in a production compiler: studying the interactions with the register coalescing and other downstream optimizations, studying load/store

optimization variants (with transparent, fine-grain live range splitting), and reducing the number of incremental allocations to compete with the slightly faster linear scan allocators.

References

[1] B. Alpern and et al. The Jikes RVM project: Building an open source research community. *IBM Systems Journal*, 44(2):399–418, 2005.

[2] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *PLDI'01*, pages 243–253, Snowbird, Utah, USA, June 2001.

[3] L. A. Belady. A study of replacement algorithms for virtual storage computers. *9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1966.

[4] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 102–114, Washington, DC, USA, 2007. IEEE Computer Society.

[5] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of spill everywhere under ssa form. In *LCTES'07*, pages 103–112, 2007.

[6] Florent Bouchez, Alain Darte, and Fabrice Rastello. Advanced conservative and optimistic register coalescing. In *CASES'08*, pages 147–156, 2008.

[7] Matthias Braun and Sebastian Hack. Register spilling and live-range splitting for ssa-form programs. In Oege de Moor and Michael Schwartzbach, editors, *Compiler Construction*, volume 5501 of *Lecture Notes in Computer Science*, pages 174–189. Springer Berlin / Heidelberg, 2009.

[8] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN'82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–105, New York, NY, 1982. ACM.

[9] Quentin Colombet, Benoit Boissinot, Philip Brisk, Sebastian Hack, and Fabrice Rastello. Graph-coloring and treescan register allocation using repairing. In *Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES'11)*, pages 45–54. IEEE Computer Society, October 2011.

[10] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *International Conference on Compiler Construction (CC'98)*, pages 174–187. Springer-Verlag, 1998.

[11] Boubacar Diouf, John Cavazos, Albert Cohen, and Fabrice Rastello. Split register allocation: Linear complexity without the performance penalty. In *Intl. Conf. on High Performance and Embedded Architectures and Compilers (HiPEAC'10)*, LNCS, Pisa, Italy, January 2010. Springer-Verlag.

[12] Janet Fabri. Automatic storage optimization. In *ACM Symp. on Compiler Construction*, pages 83–91, 1979.

[13] Martin Farach-Colton and Vincenzo Liberatore. On local register allocation. *J. of Algorithms*, 37(1):37–65, 2000.

[14] A. Frank. Some polynomial algorithms for certain graphs and hypergraphs. *Proceedings of the Fifth British Combinatorial Conference*, pages 211–226, 1975.

[15] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 2004.

[16] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *CC'06*, pages 247–262, 2006.

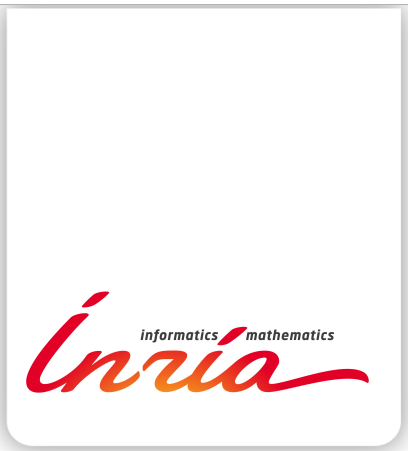
[17] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *APLAS*, pages 315–329, 2005.

[18] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999.

[19] Todd A. Proebsting and Charles N. Fischer. Probabilistic register allocation. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 300–310, New York, NY, USA, 1992. ACM.

[20] Hongbo Rong. Tree register allocation. In *International Symposium on Microarchitecture (MICRO'09)*, pages 67–77. IEEE Computer Society, December 2009.

[21] Vivek Sarkar and Rajkishore Barik. Extended linear scan: An alternate foundation for global register allocation. In Shriram Krishnamurthi and Martin Odersky, editors, *CC'07*, volume 4420 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2007.



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399